Concept Learning

Jens LEHMANN [a] and Nicola FANIZZI [b] and Lorenz BÜHMANN [a] and Claudia D'AMATO [b]

[a] *Univ. Leipzig, Germany*
[b] *Univ. Bari, Italy*

Abstract. One of the bottlenecks of the ontology construction process is the amount of work required with various figures playing a role in it: domain experts contribute their knowledge that has to be formalized by knowledge engineers so that it can be mechanized. As the gap between these roles likely makes the process slow and burdensome, this problem may be tackled by resorting to machine learning techniques. By adopting algorithms from inductive logic programming, the effort of the domain expert can be reduced, i.e. he has to label individual resources as instances of the target concept. From those labels, axioms can be induced, which can then be confirmed by the knowledge engineer. In this chapter, we survey existing methods in this area and illustrate three different algorithms in more detail. Some basics like refinement operators, decision trees and information gain are described. Finally, we briefly present implementations of those algorithms.

Keywords. Refinement operators, Decision Trees, Information Gain

1. Introduction to Concept Learning

One of the bottlenecks of the ontology construction process is represented by the amount of work required with various figures playing a role in it: domain experts contribute their knowledge that is formalized by knowledge engineers so that it can be mechanized. As the gap between these roles makes the process slow and burdensome, this problem may be tackled by resorting to *machine learning* (cf. Lawrynowicz and Tresp [23] in this volume) techniques. Solutions can be based on *relational learning* [36] which requires a limited effort from domain experts (labeling individual resources as instances of the target concepts) and leads to the construction of concepts adopting even very expressive languages [32]. If the concept learning problem is tackled as a search through a space of candidate descriptions in the reference representation guided by exemplars of the target concepts, the same algorithms can be adapted to solve also *ontology evolution* problems. Indeed, while normally the semantics of change operations has been considered from the logical and deductive point of view of automated reasoning, a relevant part of information lying in the data that populates ontological knowledge bases is generally overlooked or plays a secondary role. Early work on the application of machine learning to Description Logics (DLs) [3] essentially focused on demonstrating the PAC-learnability for various terminological languages derived from CLASSIC. In particular, Cohen and Hirsh investigate the CORECLASSIC DL proving that it is not PAC-learnable [9] as well as demonstrating the PAC-learnability of its sub-languages, such as C-CLASSIC [10], through the bottom-up LCSLEARN algorithm. These approaches tend to cast supervised concept learning to a structural generalizing operator working on equivalent graph representations of the concept descriptions. It is also worth mentioning unsupervised learning methodologies for DL concept descriptions, whose prototypical example is KLUSTER [22], a polynomial-time algorithm for the induction of BACK terminologies, which exploits the tractability of the standard inferences in this DL language [3]. More recently, approaches have been proposed that adopt the idea of *generalization as search* [33] performed through suitable operators that are specifically designed for DL languages [4,14,11,15,19,30,32] on the grounds of the previous experience in the context of ILP. There is a body of research around the analysis of such operators [30,24] along with applications to various problems [31,20] and studies on the practical scalability of algorithms using them [17,18,27]. Supervised (resp., unsupervised) learning systems, such as YINYANG [19] and DL-Learner [25], have been implemented and adoptions implemented for the ontology learning use case [7,27,8,26].

Learning alternative models such as logical decision trees offers another option for concept induction. The induction of decision trees is among the most well-known machine learning techniques [34], also in its more recent extensions that are able to work with more expressive logical representations in clausal form [5]. A new version of the FOIL algorithm [35] has been implemented, resulting in the DL-FOIL system [12]. The general framework has been extended to cope with logical representations designed for formal Web ontologies [13]. The induction of *terminological decision trees* [13], i.e. logical decision trees test-nodes represented with DL concept descriptions, adopts a classical top-down *divide-and-conquer* strategy [6] which differs from previous DL concept learning methods based on sequential covering or heuristic search, with the use of refinement operators for DL concept descriptions [19,12,32]. The main components of this new systems are 1) a set of refinement operators borrowed from other similar systems [19,31]; 2) a specific *information-gain function* which must take into account the open-world assumption, namely, many instances may be available which cannot be ascribed to the target concept nor to its negation. This requires a different setting, similar to learning with unknown class attributes [16], requiring a special treatment of the unlabeled individuals. Once a terminological tree is induced, similarly to the logical decision trees, a definition of the target concepts can be drawn exploiting the nodes in the tree structure. The algorithm has also a useful side-effect: the suggestion of new intermediate concepts which may have no definition in the current ontology.

2. Learning as Search in Description Logics

2.1. Learning Problem

In this section, the learning problem in the DL setting is formally defined.

Definition 2.1 (learning problem) *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a DL knowledge base.*

Given

- *a (new) target concept name C*

- *a set of positive and negative examples*[1] *for* C*:*
  - $*$ $\mathit{Ind}^+_C(\mathcal{A}) = \{a \in \mathit{Ind}(\mathcal{A}) \mid C(a) \in \mathcal{A}\} \subseteq R_{\mathcal{K}}(C)$ *instance retrieval of* C
  - $*$ $\mathit{Ind}^-_C(\mathcal{A}) = \{b \in \mathit{Ind}(\mathcal{A}) \mid \neg C(b) \in \mathcal{A}\} \subseteq R_{\mathcal{K}}(\neg C)$

Find *a concept definition* $C \equiv D$ *such that*

- $\mathcal{K} \models D(a) \quad \forall a \in \mathit{Ind}^+_D(\mathcal{A})$ *and*
- $\mathcal{K} \models \neg D(b) \quad \forall b \in \mathit{Ind}^-_C(\mathcal{A})$ *(resp.* $\mathcal{K} \not\models C(b) \quad \forall b \in \mathit{Ind}^-_C(\mathcal{A})$*)*

We prefer the first form for a correct definition w.r.t. negative examples ($\mathcal{K} \models \neg D(b)$) because that seems more coherent with the explicit indication given by the expert. Other settings assume ($\mathcal{K} \not\models C(b)$) which implicitly makes it a binary learning problem.

The definition given above can be interpreted as a generic supervised concept learning problem. In case a previous definition D' for C is already available in $\mathcal{K}$ and $\exists a \in \mathsf{Ind}^+_C(\mathcal{A})\ s.t.\ \mathcal{K} \not\models D'(a)$ or $\exists b \in \mathsf{Ind}^-_C(\mathcal{A})\ s.t.\ \mathcal{K} \not\models \neg D'(b)$ then the problem can be cast as a *refinement problem* which would amount to searching for a solution D starting from the approximation D'.

2.2. Refinement Operators

The solution of the learning problem stated above can be cast as a search for a correct concept definition in an ordered space $(\Sigma, \preceq)$. In such a setting, one can define suitable operators to traverse the search space. Refinement operators can be formally defined as:

Definition 2.2 (refinement operator) *Given a quasi-ordered*[2] *search space* $(\Sigma, \preceq)$

- *a* downward refinement operator *is a mapping* $\rho : \Sigma \to 2^\Sigma$ *such that*
$$\forall \alpha \in \Sigma \quad \rho(\alpha) \subseteq \{\beta \in \Sigma \mid \beta \preceq \alpha\}$$
- *an* upward refinement operator *is a mapping* $\delta : \Sigma \to 2^\Sigma$ *such that*
$$\forall \alpha \in \Sigma \quad \delta(\alpha) \subseteq \{\beta \in \Sigma \mid \alpha \preceq \beta\}$$

Definition 2.3 (properties of DL refinement operators) *A refinement operator* ρ *is*

- (locally) finite *iff* $\rho(C)$ *is finite for all concepts* C*.*
- redundant *iff there exists a refinement chain from a concept* C *to a concept* D*, which does not go through some concept* E *and a refinement chain from* C *to a concept equal to* D*, which does go through* E*.*
- proper *iff for all concepts* C *and* D*,* $D \in \rho(C)$ *implies* $C \not\equiv D$*.*

A downward refinement operator ρ *is called*

- complete *iff for all concepts* C, D *with* $C \sqsubset D$ *we can reach a concept* E *with* $E \equiv C$ *from* D *by* ρ*.*
- weakly complete *iff for all concepts* $C \sqsubset \top$ *we can reach a concept* E *with* $E \equiv C$ *from* $\top$ *by* ρ*.*

The corresponding notions for upward refinement operators are defined dually.

[1] Note that $\mathsf{Ind}^+_C(\mathcal{A}) \cup \mathsf{Ind}^-_C(\mathcal{A}) \subseteq \mathsf{Ind}(\mathcal{A})$ where $\mathsf{Ind}(\mathcal{A})$ is the set of all individuals occurring in $\mathcal{A}$.

[2] A quasi-ordering is a reflexive and transitive relation.

In the following, we will consider a space of concept definitions ordered by the subsumption relationship $\sqsubseteq$ which induces a quasi-order on the space of all the possible concept descriptions [4,11]. In particular, given the space of concept definitions in the reference DL language, say $(\mathcal{L}, \sqsubseteq)$, ordered by subsumption, there is an infinite number of generalizations and specializations. Usually one tries to devise operators that can traverse efficiently throughout the space in pursuit of one of the correct definitions (w.r.t. the examples that have been provided).

2.2.1. Refinement Operator for DL-FOIL

In the definition of refinement operators, the notion of *normal form* for $\mathcal{ALC}$ concept descriptions is given. Preliminarily, a concept is in *negation normal form* iff negation only occurs in front of concept names. Now, some notation is needed to name the different parts of an $\mathcal{ALC}$ description: $\mathsf{prim}(C)$ is the set of all the concepts at the top-level conjunction of C; if there exists a universal restriction $\forall R.D$ on the top-level of C then $\mathsf{val}_R(C) = \{D\}$ (a singleton description because many such restrictions can be collapsed into a single one with a conjunctive filler concept) otherwise $\mathsf{val}_R(C) = \{\top\}$. Finally, $\mathsf{ex}_R(C)$ is the set of the concept descriptions E appearing in existential restrictions $\exists R.E$ at the top-level conjunction of C.

Definition 2.4 ($\mathcal{ALC}$ normal form) *A concept description D is in $\mathcal{ALC}$* normal form *iff D is $\bot$ or $\top$ or if $D = D_1 \sqcup \cdots \sqcup D_n$ with*

$$D_i = \bigsqcap_{A \in \mathsf{prim}(D_i)} A \sqcap \bigsqcap_{R \in N_R} \left(\bigsqcap_{V \in \mathsf{val}_R(D_i)} \forall R.V \sqcap \bigsqcap_{E \in \mathsf{ex}_R(D_i)} \exists R.E \right)$$

where, for all $i = 1, \ldots, n$, $D_i \not\equiv \bot$ and for any R, every sub-description in $\mathsf{ex}_R(D_i)$ and $\mathsf{val}_R(D_i)$ is in normal form.

We will consider two theoretical refinement operators [19] that, given a starting incorrect definition (too weak or too strong) for the target concept in the search space, can compute one (or some) of its generalizations / specializations. Both are defined (w.l.o.g.) for $\mathcal{ALC}$ descriptions in normal form.

Definition 2.5 (downward operator ρ) *Let $\rho = (\rho_\sqcup, \rho_\sqcap)$ be a downward refinement operator, where:*
$[\rho_\sqcup]$ *given a description in $\mathcal{ALC}$ normal form $D = D_1 \sqcup \cdots \sqcup D_n$:*

- *$D' \in \rho_\sqcup(D)$ if $D' = \bigsqcup_{\substack{1 \le i \le n \\ i \neq j}} D_i$ for some $j \in \{1, \ldots, n\}$*
- *$D' \in \rho_\sqcup(D)$ if $D' = D'_i \sqcup \bigsqcup_{\substack{1 \le i \le n \\ i \neq j}} D_i$ for some $j \in \{1, \ldots, n\}$ and $D'_j \in \rho_\sqcap(D_j)$*

$[\rho_\sqcap]$ *given a conjunctive description $C = C_1 \sqcap \cdots \sqcap C_m$:*

- *$C' \in \rho_\sqcap(C)$ if $C' = C \sqcap C_{m+1}$ for some C_{m+1} such that $\bot \sqsubset C' \sqsubseteq C$*
- *$C' \in \rho_\sqcap(C)$ if $C' = \bigsqcap_{\substack{1 \le i \le m \\ i \neq k}} C_i \sqcap C'_k$ for some $k \in \{1, \ldots, m\}$, where:*

  * *$C'_k \sqsubseteq C_k$ if $C_k \in \mathsf{prim}(C)$ or*
  * *$C'_k = \exists R.D'$ if $C_k = \exists R.D$ and $D' \in \rho_\sqcup(D)$ or*

  * $C'_k = \forall R.D'$ *if* $C_k = \forall R.D$ *and* $D' \in \rho_\sqcup(D)$

Note that a difference operator for concepts is used to single out the subconcepts to be refined. Further possibilities may be explored using the operator defined $C - D = C \sqcup \neg D$ [38].

The operator for disjunctive concepts $\rho_\sqcup$ simply drops one top-level disjunct or replaces it with a downward refinement obtained with $\rho_\sqcap$. $\rho_\sqcap$ adds new conjuncts or replaces one with a refinement obtained by specializing a primitive concept or the subconcepts in the scope of a universal or existential restriction (again through $\rho_\sqcup$). Note that successive applications of the operator may require intermediate normalization steps.

Definition 2.6 (upward operator δ**)** *Let* $\delta = (\delta_\sqcup, \delta_\sqcap)$ *be a downward refinement operator, where:*
$[\delta_\sqcup]$ *given a description in* $\mathcal{ALC}$ *normal form* $D = D_1 \sqcup \cdots \sqcup D_n$*:*

- $D' \in \delta_\sqcup(D)$ *if* $D' = D \sqcup D_{n+1}$ *for some* D_{n+1} *such that* $D_{n+1} \not\sqsubseteq D$
- $D' \in \delta_\sqcup(D)$ *if* $D' = D'_j \sqcup \bigsqcup_{\substack{1 \leq i \leq n \\ i \neq j}} D_i$ *for some* $j \in \{1, \ldots .n\}$, $D'_j \in \delta_\sqcap(D_j)$

$[\delta_\sqcap]$ *given a conjunctive description* $C = C_1 \sqcap \cdots \sqcap C_m$*:*

- $C' \in \delta_\sqcap(C)$ *if* $C' = \bigsqcap_{\substack{1 \leq i \leq m \\ i \neq k}} C_i$ *for some* $k \in \{1, \ldots .m\}$
- $C' \in \delta_\sqcap(C)$ *if* $C' = \bigsqcap_{\substack{1 \leq i \leq m \\ i \neq k}} C_i \sqcap C'_k$ *for some* $k \in \{1, \ldots, m\}$*, where:*

  * $C'_k \sqsupseteq C_k$ *if* $C_k \in \mathsf{prim}(C)$ *or*
  * $C'_k = \exists R.D'$ *if* $C_k = \exists R.D$ *and* $D' \in \delta_\sqcup(D)$ *or*
  * $C'_k = \forall R.D'$ *if* $C_k = \forall R.D$ *and* $D' \in \delta_\sqcup(D)$

$\delta_\sqcup$ and $\delta_\sqcap$ simply perform dual operations w.r.t. $\rho_\sqcup$ and $\rho_\sqcap$, respectively. Some examples of their application can be found in [19].

These operators follow the definition of the $\mathcal{ALC}$ normal form. Hence they cannot be complete for more expressive DLs (see [30] for an analysis of refinement operators in DLs). However, instead of such operators that likely lead to overfit the data (e.g. a generalizing operator based on the computation of the *Least Common Subsumer* (LCS) [10] would amount to a simple union of the input descriptions in $\mathcal{ALC}$) it may be preferable to search the space (incompletely) using the non-$\mathcal{ALC}$ restrictions as atomic features (concepts). Moreover, other operators have been designed to exploit also the knowledge conveyed by the positive and negative examples in order to prune the possible candidate refinements yielded by a single generalization / specialization step and to better direct the search for suitable problem solutions [19]. Even more so, instead of using the examples in a mere *generate-and-test* strategy based on these operators, they could be exploited more directly[3], in order to influence the choices made during the refinement process.

2.2.2. *A refinement operator for CELOE*

Designing a refinement operator ρ needs to make decisions on which properties are most useful in practice regarding the underlying learning algorithm. Considering the properties *completeness*, *weak completeness*, *properness*, *finiteness*, and *non-redundancy* an

[3]E.g. using the most specific concepts [3] as their representatives to the concept level. But their exact computation is feasible only for very simple DLs.

extensive analysis in [30] has shown that the most feasible property combination for our setting is $\{\textit{weakly complete}, \textit{complete}, \textit{proper}\}$, which we will justify briefly. Only for less expressive description logics like $\mathcal{EL}$, ideal, i.e. complete, proper and final, operators exist [29]. (Weak) Completeness is considered a very important property, since an incomplete operator may fail to converge at all and thus may not return a solution even if one exists. Reasonable, weakly complete operators are often complete. Consider, for example, the situation where a weakly complete operator ρ allows to refine a concept C to $C \sqcap D$ with some $D \in \rho(\top)$. Then it turns out that this operator is already complete.

Concerning finiteness, having an infinite operator is less critical from a practical perspective since this issue can be handled algorithmically. So it is preferable not imposing finiteness, which allows to develop a proper operator. As for non-redundancy, this appears to be very difficult to achieve for more complex operators. Consider, for example, the concept $A_1 \sqcap A_2$ which can be reached from $\top$ via the chain $\top \rightsquigarrow A_1 \rightsquigarrow A_1 \sqcap A_2$, For non-redundancy, the operator would need to make sure that this concept cannot be reached via the chain $\top \rightsquigarrow A_2 \rightsquigarrow A_2 \sqcap A_1$. While there are methods to handle this in such simple cases via normal forms, it becomes more complex for arbitrarily deeply nested structures, where even applying the same replacement leads to redundancy. In the following example, A_1 is replaced by $A_1 \sqcap A_2$ twice in different order in each chain:

$$\begin{aligned}\top \rightsquigarrow & \forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.A_1 \sqcup \forall r_2.(A_1 \sqcap A_2)\\ & \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)\\ \top \rightsquigarrow & \forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.A_1\\ & \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)\end{aligned}$$

To avoid this, an operator would need to regulate when A_1 can be replaced by $A_1 \sqcap A_2$, which appears not to be achievable by syntactic replacement rules. Alternatively, a computationally inexpensive redundancy check can be used, which seems to be sufficiently useful in practice.

We now define the refinement operator ρ: For each $A \in N_C$, we define (sh stands for subsumption hierarchy):

$$\textit{sh}_{\downarrow}(A) = \{A' \in N_C \mid A' \sqsubset A, \text{ there is no } A'' \in N_C \text{ with } A' \sqsubset_{\mathcal{T}} A'' \sqsubset_{\mathcal{T}} A\}$$

$\textit{sh}_{\downarrow}(\top)$ is defined analogously for $\top$ instead of A. $\textit{sh}_{\uparrow}(A)$ is defined analogously for going upward in the subsumption hierarchy. We do the same for roles, i.e. :

$$\textit{sh}_{\downarrow}(r) = \{r' \mid r' \in N_R, r' \sqsubset r, \text{ there is no } r'' \in N_R \text{ with } r' \sqsubset_{\mathcal{T}} r'' \sqsubset_{\mathcal{T}} r\}$$

$\textit{domain}(r)$ denotes the domain of a role r and $\textit{range}(r)$ the range of a role r. A range axiom links a role to a concept. It asserts that the role fillers must be instances of a given concept. Domain axioms restrict the first argument of role assertions to a concept. We define:

$$\begin{aligned}\textit{ad}(r) = \quad & \text{an } A \text{ with } A \in \{\top\} \cup N_C \text{ and } \textit{domain}(r) \sqsubseteq A\\ & \text{and there does not exist an } A' \text{ with } \textit{domain}(r) \sqsubseteq A' \sqsubset A\end{aligned}$$

$ar(r)$ is defined analogously using *range* instead of *domain*. *ad* stands for atomic domain and *ar* stands for atomic range. We assign exactly one atomic concept as domain/range of a role. Since using atomic concepts as domain and range is very common, *domain* and *ad* as well as *range* and *ar* will usually coincide. The set app_B of applicable properties with respect to an atomic concept B is defined as:

$$app_B = \{r | r \in N_R, ad(r) = A, A \sqcap B \not\equiv \bot\}$$

To give an example, for the concept `Person`, we have that the role `hasChild` with $ad(\texttt{hasChild}) = \texttt{Person}$ is applicable, but the role `hasAtom` with $ad(\texttt{hasAtom}) = \texttt{ChemicalCompound}$ is not applicable (assuming `Person` and `ChemicalCompound` are disjoint). We will use this to restrict the search space by ruling out unsatisfiable concepts. The index B describes the context in which the operator is applied, e.g. $\top \rightsquigarrow \texttt{Person}$ is a refinement step of ρ. However, $\exists\texttt{hasAtom}.\top \rightsquigarrow \exists\texttt{hasAtom}.\texttt{Person}$ is not a refinement step of ρ assuming $ar(\texttt{hasAtom})$ and `Person` are disjoint. The set of most general applicable roles mgr_B with respect to a concept B is defined as:

$$mgr_B = \{r \mid r \in app_B, \text{ there is no } r' \text{ with } r \sqsubset r', r' \in app_B\}$$

M_B with $B \in \{\top\} \cup N_C$ is defined as the union of the following sets:

- $\{A \mid A \in N_C, A \sqcap B \not\equiv \bot, A \sqcap B \not\equiv B, \text{ there is no } A' \in N_C \text{ with } A \sqsubset A'\}$
- $\{\neg A \mid A \in N_C, \neg A \sqcap B \not\equiv \bot, \neg A \sqcap B \not\equiv B, \text{ there is no } A' \in N_C \text{ with } A' \sqsubset A\}$
- $\{\exists r.\top \mid r \in mgr_B\}$
- $\{\forall r.\top \mid r \in mgr_B\}$

The operator ρ is defined in Figure 1. Note that ρ delegates to an operator ρ_B with $B = \top$ initially. B is set to the atomic range of roles contained in the input concept when the operator recursively traverses the structure of the concept. The index B in the operator (and the set M above) is used to rule out concepts which are disjoint with B.

Example 2.1 (ρ refinements) *Since the operator is not easy to understand at first glance, we provide some examples. Let the following knowledge base be given:*

$$\mathcal{K} = \{\mathit{Man} \sqsubset \mathit{Person}; \mathit{Woman} \sqsubset \mathit{Person}; \mathit{SUV} \sqsubset \mathit{Car}; \mathit{Limo} \sqsubset \mathit{Car};$$
$$\mathit{Person} \sqcap \mathit{Car} \equiv \bot; \mathit{domain}(\mathit{hasOwner}) = \mathit{Car}; \mathit{range}(\mathit{hasOwner}) = \mathit{Person}\}$$

Then the following refinements of $\top$ exist:

$$\rho(\top) = \{\mathit{Car}, \mathit{Person}, \neg\mathit{Limo}, \neg\mathit{SUV}, \neg\mathit{Woman}, \neg\mathit{Man},$$
$$\exists\mathit{hasOwner}.\top, \forall\mathit{hasOwner}.\top, \mathit{Car} \sqcup \mathit{Car}, \mathit{Car} \sqcup \mathit{Person}, \dots\}$$

This illustrates how the set $M_\top$ is constructed. Note that refinements like $\mathit{Car} \sqcup \mathit{Car}$ are incorporated in order to reach e.g. $\mathit{SUV} \sqcup \mathit{Limo}$ later in a possible refinement chain. The concept $\mathit{Car} \sqcap \exists\mathit{hasOwner}.\mathit{Person}$ has the following refinements:

$$\rho(C) = \begin{cases} \{\bot\} \cup \rho_\top(C) & \text{if } C = \top \\ \rho_\top(C) & \text{otherwise} \end{cases}$$

$$\rho_B(C) = \begin{cases} \emptyset & \text{if } C = \bot \\ \{C_1 \sqcup \dots \sqcup C_n \mid C_i \in M_B \ (1 \le i \le n)\} & \text{if } C = \top \\ \{A' \mid A' \in sh_\downarrow(A)\} & \text{if } C = A \ (A \in N_C) \\ \quad \cup \{A \sqcap D \mid D \in \rho_B(\top)\} & \\ \{\neg A' \mid A' \in sh_\uparrow(A)\} & \text{if } C = \neg A \ (A \in N_C) \\ \quad \cup \{\neg A \sqcap D \mid D \in \rho_B(\top)\} & \\ \{\exists r.E \mid A = ar(r), E \in \rho_A(D)\} & \text{if } C = \exists r.D \\ \quad \cup \{\exists r.D \sqcap E \mid E \in \rho_B(\top)\} & \\ \quad \cup \{\exists s.D \mid s \in sh_\downarrow(r)\} & \\ \{\forall r.E \mid A = ar(r), E \in \rho_A(D)\} & \text{if } C = \forall r.D \\ \quad \cup \{\forall r.D \sqcap E \mid E \in \rho_B(\top)\} & \\ \quad \cup \{\forall r.\bot \mid & \\ \qquad D = A \in N_C \text{ and } sh_\downarrow(A) = \emptyset\} & \\ \quad \cup \{\forall s.D \mid s \in sh_\downarrow(r)\} & \\ \{C_1 \sqcap \dots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \dots \sqcap C_n \mid & \text{if } C = C_1 \sqcap \dots \sqcap C_n \\ \qquad D \in \rho_B(C_i), 1 \le i \le n\} & (n \ge 2) \\ \{C_1 \sqcup \dots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \dots \sqcup C_n \mid & \text{if } C = C_1 \sqcup \dots \sqcup C_n \\ \qquad D \in \rho_B(C_i), 1 \le i \le n\} & (n \ge 2) \\ \quad \cup \{(C_1 \sqcup \dots \sqcup C_n) \sqcap D \mid & \\ \qquad D \in \rho_B(\top)\} & \end{cases}$$

Figure 1. Definition of the refinement operator ρ.

$$\begin{aligned} \rho(\mathtt{Car} \sqcap \exists \mathtt{hasOwner.Person}) = \{ & \mathtt{Car} \sqcap \exists \mathtt{hasOwner.Man}, \\ & \mathtt{Car} \sqcap \exists \mathtt{hasOwner.Woman}, \\ & \mathtt{SUV} \sqcap \exists \mathtt{hasOwner.Person}, \\ & \mathtt{Limo} \sqcap \exists \mathtt{hasOwner.Person}, \dots \} \end{aligned}$$

Note the traversal of the subsumption hierarchy, e.g. `Car` *is replaced by* `SUV`*.*

Proposition 2.1 (Downward Refinement of ρ) *ρ is an $\mathcal{ALC}$ downward refinement operator.*

A distinguishing feature of ρ compared to other DL refinement operators [4,11], is that it makes use of the subsumption and role hierarchy, e.g. for concepts $A_2 \sqsubset A_1$, we reach A_2 via $\top \rightsquigarrow A_1 \rightsquigarrow A_2$. This way, we can stop the search if A_1 is already too weak and, thus, make better use of TBox knowledge. The operator also uses domain and range of roles to reduce the search space. This is similar to mode declarations in Aleph, Progol, and other ILP programs. However, in DL knowledge bases and OWL ontologies, domain and range are usually explicitly given, so there is no need to define them manually. Overall, the operator supports more structures than those in [4,11] and tries to intelligently incorporate background knowledge. In [32] further extensions of the operator are described, which increase its expressivity such that it can handle most OWL class expressions. Note that ρ is infinite. The reason is that the set M_B is infinite and we put no bound on the number of elements in the disjunctions, which are refinements of the top concept. Furthermore, the operator requires reasoner requests for calculating M_B. However, the number of requests is fixed, so – assuming the results of those requests are cached – the reasoner is only needed in an initial phase, i.e. during the first calls to the refinement operator. This means that, apart from this initial phase, the refinement operator performs only syntactic rewriting rules.

3. CELOE

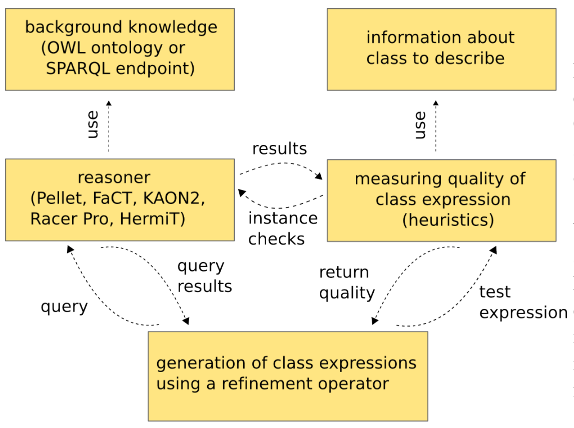

background knowledge (OWL ontology or SPARQL endpoint)
information about class to describe
use
use
results
reasoner (Pellet, FaCT, KAON2, Racer Pro, HermiT)
measuring quality of class expression (heuristics)
instance checks
query results
return quality
query
test expression
generation of class expressions using a refinement operator

Figure 2. Outline of the general learning approach in CELOE: One part of the algorithm is the generation of promising class expressions taking the available background knowledge into account. Another part is a heuristic measure of how close an expression is to being a solution of the learning problem. Figure adapted from [17,18].

Figure 2 gives an overview of our algorithm *CELOE* (standing for “class expression learning for ontology engineering”), which follows the common “generate and test“ approach in ILP. Learning is seen as a search process and several class expressions are generated and tested against a background knowledge base. Each of those class expressions is evaluated using a heuristic [27]. A challenging part of a learning algorithm is to decide which expressions to test. Such a decision should take the computed heuristic values and the structure of the background knowledge into account. For CELOE, we use the approach described in [31,32] as base, where this problem has been analysed, implemented, and evaluated. It is based on the *refinement operator* introduced in Sect. 2.2.2.

The approach we used is a top-down algorithm based on refinement operators as illustrated in Figure 3. This means that the first class expression, which will be tested is the most general expression ($\top$), which is then mapped to a set of more specific expressions by means of a downward refinement operator. The refinement operator can be applied to the obtained expressions again, thereby spanning a *search tree*. The search tree can be pruned when an expression does not cover sufficiently many instances of the class A we want to describe. One example for a path in a search tree spanned up by a downward refinement operator is the following ($\rightsquigarrow$ denotes a refinement step):

$$\top \rightsquigarrow \texttt{Person} \rightsquigarrow \texttt{Person} \sqcap \texttt{takesPartinIn}.\top$$
$$\rightsquigarrow \texttt{Person} \sqcap \texttt{takesPartIn.Meeting}$$

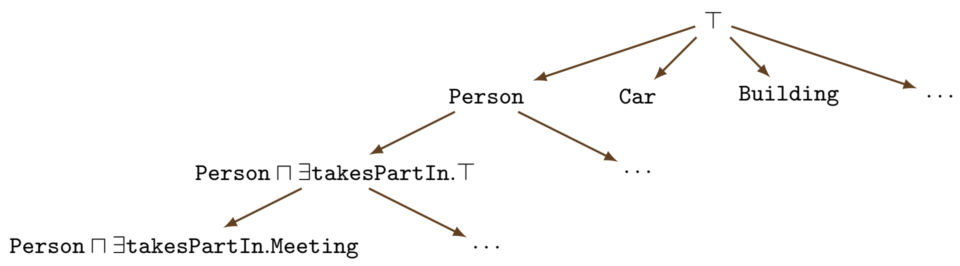


Figure 3. Illustration of a search tree in a top down refinement approach.

The heart of such a learning strategy is to define a suitable refinement operator and an appropriate search heuristics for deciding which nodes in the search tree should be expanded. The refinement operator in the considered algorithm is defined in [32]. It is based on [31] which in turn is build on theoretical foundations in [30]. It has been shown to be the best achievable operator with respect to a set of properties (not further described here), which are used to assess the performance of refinement operators. The learning algorithm supports conjunction, disjunction, negation, existential and universal quantifiers, cardinality restrictions, hasValue restrictions as well as boolean and double datatypes.

While the major change compared to other supervised learning algorithms for OWL is the previously described heuristic, there are also further modifications. The goal of those changes is to adapt the learning algorithm to the ontology engineering scenario: For example, the algorithm was modified to introduce a strong bias towards short class expressions. This means that the algorithm is less likely to produce long class expressions, but is almost guaranteed to find any suitable short expression (see [8] for an alternative approach to achieve this). The rationale behind this change is that knowledge engineers can understand short expressions better than more complex ones and it is essential not to miss those. We also introduced improvements to enhance the readability of suggestions: Each suggestion is reduced, i.e. there is a guarantee that they are as succinct as possible. For example, $\exists \texttt{hasLeader}.\top \sqcap \texttt{Capital}$ is reduced to $\texttt{Capital}$ if the background knowledge allows to infer that a capital is a city and each city has a leader. This reduction algorithm uses the complete and sound *Pellet* reasoner, i.e. it can take any possible complex relationships into account by performing a series of subsumption checks between class expressions. A caching mechanism is used to store the results of those checks, which allows to perform the reduction very efficiently after a warm-up phase. We also make sure that "redundant" suggestions are omitted. If one suggestion is longer and subsumed by another suggestion and both have the same characteristics, i.e. classify the relevant individuals equally, the more specific suggestion is filtered. This avoids expressions containing irrelevant subexpressions and ensures that the suggestions are sufficiently diverse.

4. DL-Foil

In this section, DL-Foil [12] algorithm is presented. The main aim of this work was conceiving a learning algorithm that could overcome two limitation of the current DL concept learning systems, namely avoiding the computation of the most specific concepts (which is also language-dependent) and the excessive (syntactic) complexity of the

Algorithm 1 GENERALIZE(*Positives*, *Negatives*, *Unlabeled*): *Generalization*

```
Require: Positives, Negatives, Unlabeled: positive, negative and unlabeled individuals
Ensure: Generalization: concept definition solving the learning problem
 1: Generalization ← $\bot$
 2: PositivesToCover ← Positives
 3: while PositivesToCover $\neq \emptyset$ do
 4:    PartialDef ← $\top$
 5:    CoveredNegatives ← Negatives
 6:    while CoveredNegatives $\neq \emptyset$ do
 7:       PartialDef ← SPECIALIZE(PartialDef, PositivesToCover, CoveredNegatives, Unlabeled)
 8:       CoveredNegatives ← $\{n \in$ Negatives $\mid \mathcal{K} \models \neg$PartialDef$(n)\}$
 9:    end while
10:    CoveredPositives ← $\{p \in$ PositivesToCover $\mid \mathcal{K} \models$ PartialDef$(p)\}$
11:    Generalization ← Generalization $\sqcup$ PartialDef
12:    PositivesToCover ← PositivesToCover $\setminus$ CoveredPositives
13: end while
14: return Generalization
```

resulting generalizations. For instance, the algorithm presented in [19] requires lifting the instances to the concept level through a suitable approximate MSC operator and then start learning from such extremely specific concept descriptions. This setting has the disadvantages of approximation and language-dependency. In DL-LEARNER [31] these drawbacks are partly mitigated because a learning procedure grounded on a genetic programming based on refinement operators is adopted, whose fitness is computed on the grounds of the covered instances (retrieval). More heuristics and approximated retrieval procedures are further investigated in [17].

The DL-FOIL algorithm essentially adapts the original FOIL algorithm [35] to the different learning problem with DL knowledge bases. Together with a sequential covering procedure, it exploits the (downward) refinement operators defined in Sect. 2.2.1 and a heuristic similar to the *information gain* to select among candidate specialization. Various search strategies have been experimented as well as evaluation measures. Those that we will present in the following are those which gave the best results. A sketch of the main routine of the learning procedure is reported as Alg. 1. Like in the original FOIL algorithm, the generalization routine computes (partial) generalizations as long as they do not cover any negative example. If this occurs, the specialization routine is invoked for solving these sub-problems. This routine applies the idea of specializing using the (incomplete) refinement operator defined in the previous section. The specialization continues until no negative example is covered (or a very limited amount[4] of them). The partial generalizations built on each outer loop are finally grouped together in a disjunction which is an allowed constructor for more expressive logics than (or equal to) $\mathcal{ALC}$. Also the outer while-loop can be exited before covering all the positive examples for avoiding overfitting generalizations.

The specialization function SPECIALIZE (reported as Alg. 2) is called from within the inner loop of the generalization procedure in order to specialize an overly general partial generalization. The function searches for proper refinements that provide at least a minimal gain (see below) fixed with a threshold (*MINGAIN*). Specializations are ran-

[4]The actual exit-condition for the inner loop may be: $1 - |\mathit{CoveredNegatives}|/|\mathit{Negatives}| < \varepsilon$, for some small constant ε.

Algorithm 2 SPECIALIZE(*PartialDef*, *Positives*, *Negatives*, *Unlabeled*): *Refinement*

```
Require:
    PartialDef: concept definition
    Positives, Negatives, Unlabeled: (positive, negative and unlabeled) individuals
Ensure: Refinement: concept definition
 1: const:
    MINGAIN: minimal acceptable gain;
    NUMSPECS: number of specializations to be generated
 2: bestGain ← 0
 3: while bestGain < MINGAIN do
 4:   for $i \leftarrow 1$ to NUMSPECS do
 5:     Specialization ← GETRANDOMREFINEMENT($\rho$, PartialDef)
 6:     CoveredPositives ← $\{p \in$ Positives $\mid \mathcal{K} \models$ Specialization$(p)\}$
 7:     CoveredNegatives ← $\{n \in$ Negatives $\mid \mathcal{K} \models \neg$Specialization$(n)\}$
 8:     thisGain ← GAIN(CoveredPositives, CoveredNegatives, Unlabeled, Positives, Negatives)
 9:     if thisGain > bestGain then
10:       bestConcept ← Specialization
11:       bestGain ← thisGain
12:     end if
13:   end for
14: end while
15: return Refinement
```

domly generated using the ρ operator defined in Sect. 2.2.1, especially $\rho_\sqcap$ is exploited with the addition of new conjuncts or the specialization of primitive concepts or role restrictions. This is similar to the original FOIL algorithm, where new random literals are appended to clauses' antecedents. A first random choice is made between atomic concepts or role restrictions. In the latter case another random choice is made between existential and universal restriction. In all cases the required concept and roles names are also randomly selected. This may give a way to impose some further bias to the form of the concept descriptions to be induced.

As regards the heuristic employed to guide the search, it was shown [21] that the gain function has to take into account incomplete examples. Similarly to a semi-supervised learning setting, the gain value g that is computed in GAIN() for selecting the best refinement is obtained as follows:

$$g = p_1 \cdot \left[\log \frac{p_1 + u_1 w_1}{p_1 + n_1 + u_1} - \log \frac{p_0 + u_0 w_0}{p_0 + n_0 + u_0}\right]$$

where p_1, n_1 and u_1 represent, resp., the number of positive, negative and unlabeled examples covered by the specialization; p_0, n_0 and u_0 stand for the number of positive, negative and unlabeled examples covered by the former definition, the weights w_0, w_1 can be determined by an estimate of the prior probability of the positive examples, resp., in the current and former concept definition. To avoid the case of null numerators, a further correction of the probabilities is performed by resorting to an m-estimate procedure.

The overall complexity of the algorithm is largely determined by the calls to reasoning services, namely subsumption (satisfiability) and instance-checking. If we consider only concepts expressed in $\mathcal{ALC}$ logic, the complexity of these inferences is P-space. However, should the considered knowledge base contain definitions expressed in more

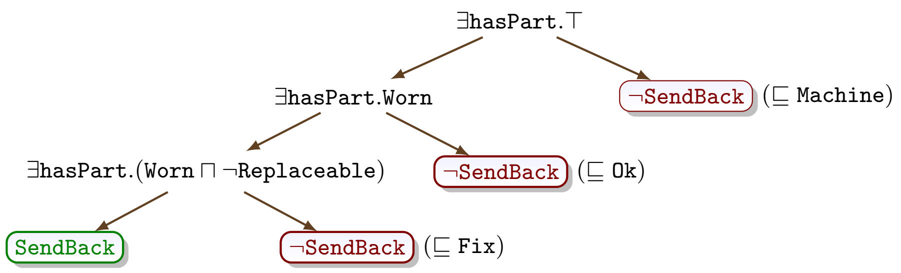


Figure 4. A TDT whose leftmost path corresponds to the DL concept definition SendBack ≡ ∃hasPart.(Worn ⊓ ¬Replaceable). Other definitions can be associated to the paths to leaves labeled with ¬SendBack that are related to other (disjoint) concepts.

complex languages which require more complex reasoning procedures. The inductive algorithm can be thought as a means for building (upper) $\mathcal{ALC}$-approximations of the target concepts. The number of nodes visited during the traversal of the search space grows with richness of the vocabulary, yet not with the expressiveness of the underlying DL, because the algorithm searches a sub-space of the actual search space induced by the language.

5. Learning Terminological Decision Trees

First-order logical decision trees (FOLDTs) [5] are binary decision trees in which

1. the nodes contain tests in the form of conjunctions of literals;
2. left and right branches stand, resp., for the truth-value (resp. true and false) determined by the test evaluation;
3. different nodes may share variables, yet a variable that is introduced in a certain node must not occur in the right branch of that node.

Terminological decision trees (TDTs) extend the original definition, allowing DL concept descriptions as (variable-free) node tests. Fig. 4 shows a TDT denoting also the definition of the SendBack concept as in the problem described in [5].

5.1. Classification

The TDTs can be used for classifying individuals. Alg. 3 shows the related classification procedure. It uses other functions: LEAF() to determine whether a node is a leaf of the argument tree, ROOT() which returns the root node of the input tree, and INODE() which retrieves the test concept and the left and right subtrees branching from a given internal node. Given an individual a, starting from the root node, the algorithm checks the class-membership w.r.t. the test concept D_i in the current node, i.e. $\mathcal{K} \models D_i(a)$, sorting a to the left branch if the test is successful while the right branch is chosen if $\mathcal{K} \models \neg D_i(a)$. Eventually the classification is found as a leaf-node concept.

Note that the open-world semantics may cause unknown answers (failure of both left and right branch tests) that can be avoided by considering a weaker (default) right-branch test: $\mathcal{K} \not\models D_i(a)$. This differs from the FOLDTs where the test actually consists of several conjunctions that occur in the path from the root to the current node.

Algorithm 3 Classification with TDTs.

```
CLASSIFY($a$: individual, $T$: TDT, $\mathcal{K}$: KB): concept;
1. $N \leftarrow$ ROOT($T$);
2. while $\neg$LEAF($N, T$) do
    (a) $(D, T_{\text{left}}, T_{\text{right}}) \leftarrow$ INODE($N$);
    (b) if $\mathcal{K} \models D(a)$ then $N \leftarrow$ ROOT($T_{\text{left}}$)
    (c) elseif $\mathcal{K} \models \neg D(a)$ then $N \leftarrow$ ROOT($T_{\text{right}}$)
    (d) else return $\top$
3. $(D, \cdot, \cdot) \leftarrow$ INODE($N$);
4. return $D$;
```

5.2. From Terminological Decision Trees to Concept Descriptions

Note that each node in a path may be used to build a concept description through specializations. This can be given 1) by adding a conjunctive concept description, 2) by refining a sub-description in the scope of an existential, universal or number restriction or 3) by narrowing a number restriction (which may be allowed by the underlying language, e.g. $\mathcal{ALN}$ or $\mathcal{ALCQ}$). No special care[5] is to be devoted to negated atoms and their variables.

For each target concept name C it is possible to derive a *single* concept definition from a TDT. The algorithm (see Alg. 4) follows all the paths leading to success nodes i.e. leaves labeled with C (the heads of the clauses in the original setting) collecting the intermediate test concepts (formerly, the body literals). In this way, each path yields a different conjunctive concept description that represents a different version of the target concept in conjunctive form $D_i = D_1^i \sqcap \cdots \sqcap D_l^i$. The final single description for the target concept is obtained as the disjunctive description built with concepts from this finite set $S = \{D_i\}_{i=1}^M$. Hence, the final definition is $C \equiv \bigsqcup_{i=1}^M D_i$. As an example, looking at the TDT depicted in Figure 4, a concept definition that may be extracted is

$$\texttt{Ok} \equiv \exists\texttt{hasPart}.\top \sqcap \neg\exists\texttt{hasPart.Worn} \equiv \exists\texttt{hasPart}.\top \sqcap \forall\texttt{hasPart}.\neg\texttt{Worn}$$

i.e. something that has exclusively parts which are not worn.

Like in the original logic tree induction setting, also internal nodes may be utilized to induce new intermediate concepts.

5.3. Induction of TDTs

The subsumption relationship $\sqsubseteq$ induces a partial order on the space of DL concept descriptions. Then, as seen above, the learning task can be cast as a search for a solution of the problem in the partially ordered space. In such a setting, suitable operators to traverse the search space are required [19,32]. While existing DL concept induction algorithms generally adopt a separate-and-conquer covering strategy, the TDT-learning algorithm adopts a divide-and-conquer strategy [6]. It also tries to cope with the limitations of the other learning systems, namely approximation and language-dependence. Indeed, since the early works [10], instances are required to be transposed to the concept level before the learning can start. This is accomplished by resorting to the computation, for each training individual, of the related MSC the individual belongs to [3], which need not ex-

[5]We are considering expressive (and decidable) DL languages like $\mathcal{ALCQ}$, that are endowed with full negation, hence the situation is perfectly symmetric.

Algorithm 4 Mapping a TDT onto a DL concept description.

```
DERIVEDEFINITION($C$: concept name, $T$: TDT): concept description;

1. $S \leftarrow$ ASSOCIATE$(C, T, \top)$;
2. return $\bigsqcup_{D \in S} D$;

ASSOCIATE($C$: concept name; $T$: TDT; $D_c$: current concept description): set of descriptions;

1. $N \leftarrow$ ROOT$(T)$;
2. $(D_n, T_{\text{left}}, T_{\text{right}}) \leftarrow$ INODE$(N)$;
3. if LEAF$(N, T)$ then
     (a) if $D_n = C$ then
             return $\{D_c\}$;
         else
             return $\emptyset$;
   else
     (a) $S_{\text{left}} \leftarrow$ ASSOCIATE$(C, T_{\text{left}}, D_c \sqcap D_n)$;
     (b) $S_{\text{right}} \leftarrow$ ASSOCIATE$(C, T_{\text{right}}, D_c \sqcap \neg D_n)$;
     (c) return $S_{\text{left}} \cup S_{\text{right}}$;
```

ist, especially for expressive DLs, and thus has to be approximated. Even in an approximated version, the MSCs turn out to be extremely specific descriptions which affects both the efficiency of learning and the effectiveness of the learned descriptions as this specificity easily leads to overfitting the data [19].

The algorithms implemented by DL-LEARNER [32] partly mitigate these disadvantages being based on stochastic search using refinement operators and a heuristic computed on the grounds of the covered individuals (and a syntactic notion of concept length). Generate-and-test strategies may fall short when considering growing search spaces determined by more expressive languages. This drawback is hardly avoidable and it has been tackled by allowing more interaction with the knowledge engineer which can be presented with partial solutions and then decide to stop further refinements.

Our TDT-induction algorithm adapts the classic schema implemented by C4.5 [34] and TILDE [5]. A sketch of the main routine is reported as Alg. 5. It reflects the standard tree induction algorithms with the addition of the treatment of unlabeled training individuals. The three initial conditions take care of the base cases of the recursion, namely:

1. no individuals got sorted to the current subtree root then the resulting leaf is decided on the grounds of the prior probabilities of positive and negative instances (resp. Pr_+ and Pr_-);
2. no negative individual yet a sufficient rate (w.r.t. the threshold θ) of positive ones got sorted to the current node, then the leaf is labeled accordingly;
3. dual case w.r.t. to the previous one.

The second half of the algorithm (randomly) generates a set *Specs* of (satisfiable) candidate descriptions (calling GENERATENEWCONCEPTS), that can specialize the current description D when added as a conjunction. Then, the best one (D_{best}) is selected in terms of an improvement of the purity of the subsets of individuals resulting from a split based on the test description. The (im)purity measure is based on the entropic *infor-*

Algorithm 5 The main routine for inducing terminological decision trees
INDUCETDTREE(C: concept name; D: current description; *Ps*, *Ns*, *Us*: set of (positive, negative, unlabeled) training individuals): TDT;

```
1: const $\theta$;                                         {purity threshold}
2: Initialize new TDT $T$;
3: if $|Ps| = 0$ and $|Ns| = 0$ then
4:     if $Pr_{+} \geq Pr_{-}$ then
5:         $T$.root $\leftarrow C$
6:     else
7:         $T$.root $\leftarrow \neg C$;
8:     end if
9:     return $T$;
10: end if
11: if $|Ns| = 0$ and $|Ps|/(|Ps| + |Us|) > \theta$ then
12:     $T$.root $\leftarrow C$; return $T$;
13: end if
14: if $|Ps| = 0$ and $|Ns|/(|Ns| + |Us|) > \theta$ then
15:     $T$.root $\leftarrow \neg C$; return $T$;
16: end if
17: $Specs \leftarrow$ GENERATENEWCONCEPTS($D, Ps, Ns$);
18: $D_{best} \leftarrow$ SELECTBESTCONCEPT($Specs, Ps, Ns, Us$);
19: $((P^l, N^l, U^l), (P^r, N^r, U^r)) \leftarrow$ SPLIT($D_{best}, Ps, Ns, Us$);
20: $T$.root $\leftarrow D_{best}$;
21: $T$.left $\leftarrow$ INDUCETDTREE($C, D \sqcap D_{best}, P^l, N^l, U^l$);
22: $T$.right $\leftarrow$ INDUCETDTREE($C, D \sqcap \neg D_{best}, P^r, N^r, U^r$);
23: return $T$;
```

mation gain [34] or on the *Gini index* which was finally preferred. In the DL setting the problem is made more complex by the presence of instances which cannot be labelled as positive or negative (see [12]) whose contributions are considered as proportional to the prior distribution of positive and negative examples.

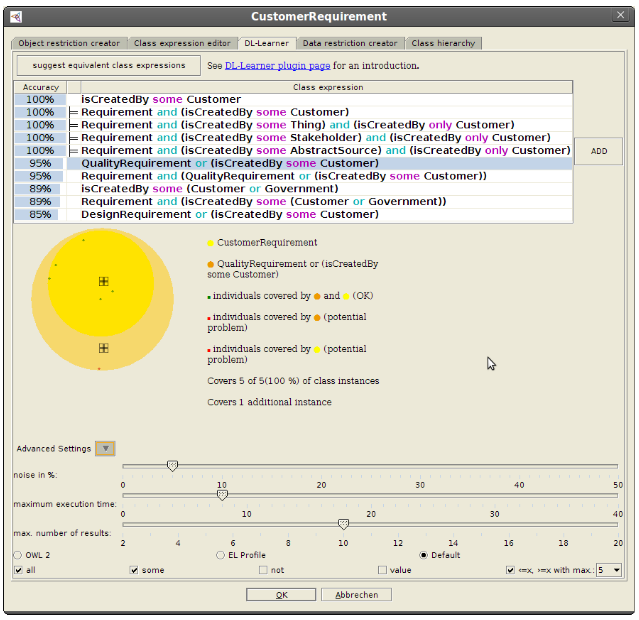
Once the best description D_{best} has been selected (calling SELECTBESTCONCEPT), it is installed as the current subtree root and the sets of individuals sorted to this node are subdivided according to their classification w.r.t. such a concept. Note that unlabeled individuals must be sorted to both subtrees. Finally the recursive calls for the construction of the subtrees are made, passing the proper sets of individuals and the concept descriptions $D \sqcap D_{best}$ and $D \sqcap \neg D_{best}$ related to either path.

The resulting system, TERMITIS (TERM*inological* T*ree* I*nduction* S*ystem*), ver. 1.2, was applied, for comparative purposes, to ontologies that have been considered in previous experiments with other DL learning systems [13].

6. Implementation

6.1. The Protégé Plugin

After implementing and testing the learning algorithm described in Sect. 3, it has been integrated into *Protégé* and *OntoWiki*. We extended the Protégé 4 plugin mechanism to be able to integrate the DL-Learner plugin as an additional method to create class



See DL-Learner plugin page for an introduction.

Accuracy		Class expression
100%		isCreatedBy some Customer
100%	⊨	Requirement and (isCreatedBy some Customer)
100%	⊨	Requirement and (isCreatedBy some Thing) and (isCreatedBy only Customer)
100%	⊨	Requirement and (isCreatedBy some Stakeholder) and (isCreatedBy only Customer)
100%	⊨	Requirement and (isCreatedBy some AbstractSource) and (isCreatedBy only Customer)
95%		QualityRequirement or (isCreatedBy some Customer)
95%		Requirement and (QualityRequirement or (isCreatedBy some Customer))
89%		isCreatedBy some (Customer or Government)
89%		Requirement and (isCreatedBy some (Customer or Government))
85%		DesignRequirement or (isCreatedBy some Customer)

● CustomerRequirement

● QualityRequirement or (isCreatedBy some Customer)

▪ individuals covered by ● and ● (OK)

▪ individuals covered by ● (potential problem)

▪ individuals covered by ● (potential problem)

Covers 5 of 5(100 %) of class instances

Covers 1 additional instance

noise in %: 0 10 20 30 40 50

maximum execution time: 0 10 20 30 40

max. number of results: 2 4 6 8 10 12 14 16 18 20

OWL 2 EL Profile [✓ Default]

[✓ all] [✓ some] not value [✓ <=x, >=x with max.: 5]

Figure 5. A screenshot of the DL-Learner Protégé plugin. It is integrated as additional tab to create class expressions in Protégé. The user is only required to press the "suggest equivalent class expressions" button and within a few seconds they will be displayed ordered by accuracy. If desired, the knowledge engineer can visualize the instances of the expression to detect potential problems. At the bottom, optional expert configuration settings can be adopted.

expressions. The plugin has also become part of the official Protégé 4 repository. A screenshot of the plugin is shown in Fig. 5. To use the plugin, the knowledge engineer is only required to press a button, which then starts a new thread, in the background, that executes the learning algorithm. The used algorithm is an *anytime algorithm*, i.e. at each point in time we can always see the currently best suggestions. The GUI updates the suggestion list each second until the maximum runtime – 10 seconds per default – is reached. For each suggestion, the plugin displays its accuracy. When clicking on a suggestion, it is visualized by displaying two circles: One stands for the instances of the class to describe and another circle for the instances of the suggested class expression. Ideally, both circles overlap completely, but in practice this will often not be the case. Clicking on the plus symbol in each circle shows its list of individuals. Those individuals are also presented as points in the circles and moving the mouse over such a point shows information about the respective individual. Red points show potential problems, where it is important to note that we use a closed world assumption to detect those. If there is not only a potential problem, but adding the expression would render the ontology inconsistent, the suggestion is marked red and a warning message is displayed. Accepting such a suggestion can still be a good choice, because the problem often lies elsewhere in the knowledge base, but was not obvious before, since the ontology was not sufficiently expressive for reasoners to detect it. This is illustrated by a screencast available from the plugin homepage,[6] where the ontology becomes inconsistent after adding the axiom, and the real source of the problem is fixed afterwards. Being able to make such suggestions can be seen as a strength of the plugin.

The plugin allows the knowledge engineer to change expert settings. Those settings include the maximum suggestion search time, the number of results returned and settings related to the desired target language., e.g. the knowledge engineer can choose to stay

[6]`http://dl-learner.org/wiki/ProtegePlugin`

within the OWL 2 EL profile or enable/disable certain class expression constructors. The learning algorithm is designed to be able to handle noisy data and the visualisation of the suggestions will reveal false class assignments so that they can be fixed afterwards.

6.2. The OntoWiki Plugin

Analogous to Protégé, we created a similar plugin for OntoWiki [2,1]. OntoWiki is a lightweight ontology editor, which allows distributed and collaborative editing of knowledge bases. The DL-Learner plugin is technically realized by implementing an OntoWiki component, which contains the core functionality, and a module, which implements the UI embedding. The DL-Learner plugin can be invoked from several places in OntoWiki, for instance through the context menu of classes. The plugin accesses DL-Learner functionality through its WSDL-based web service interface. Jar files containing all necessary libraries are provided by the plugin. If a user invokes the plugin, it scans whether the web service is online at its default address. If not, it is started automatically.

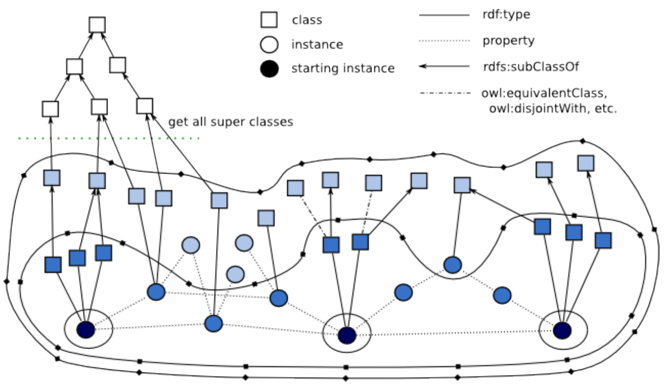
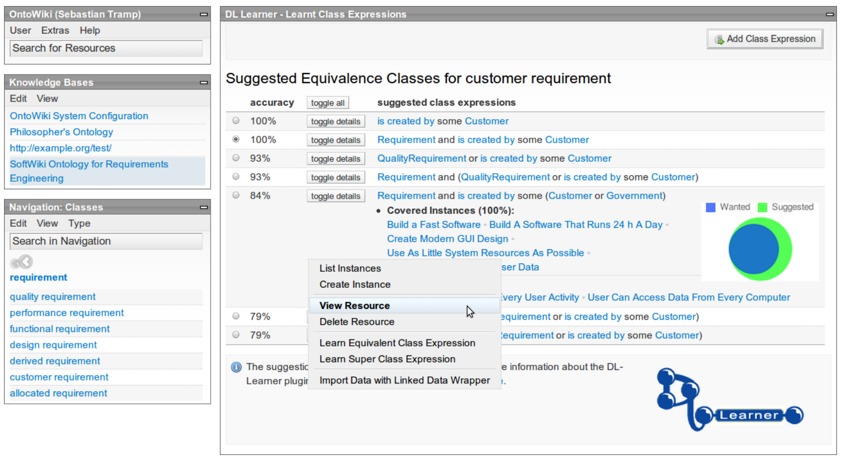


Figure 6. Extraction with three starting instances. The circles represent different recursion depths. The circles around the starting instances signify recursion depth 0. The larger inner circle represents the fragment with recursion depth 1 and the largest outer circle with recursion depth 2. Figure taken from [17].

A major technical difference compared to the Protégé plugin is that the knowledge base is accessed via SPARQL, since OntoWiki is a SPARQL-based web application. In Protégé, the current state of the knowledge base is stored in memory in a Java object. As a result, we cannot easily apply a reasoner on an OntoWiki knowledge base. To overcome this problem, we use the DL-Learner fragment selection mechanism described in [17]. Starting from a set of instances, the mechanism extracts a relevant fragment from the underlying knowledge base up to some specified recursion depth. Fig. 6 provides an overview of the fragment selection process. The fragment has the property that learning results on it are similar to those on the complete knowledge base. For a detailed description see [17]. The fragment selection is only performed for medium to large-sized knowledge bases. Small knowledge bases are retrieved completely and loaded into the reasoner. While the fragment selection can cause a delay of several seconds before the learning algorithm starts, it also offers flexibility and scalability. For instance, we can learn class expressions in large knowledge bases such as DBpedia in OntoWiki. Fig. 7 shows a screenshot of the OntoWiki plugin applied to the SWORE [37] ontology. Sug-



Suggested Equivalence Classes for customer requirement

accuracy		suggested class expressions
100%	toggle details	is created by some Customer
100%	toggle details	Requirement and is created by some Customer
93%	toggle details	QualityRequirement or is created by some Customer
93%	toggle details	Requirement and (QualityRequirement or is created by some Customer)
84%	toggle details	Requirement and is created by some (Customer or Government) • Covered Instances (100%): Build a Fast Software • Build A Software That Runs 24 h A Day • Create Modern GUI Design • Use As Little System Resources As Possible • ser Data … very User Activity • User Can Access Data From Every Computer
79%		equirement or is created by some Customer)
79%		equirement or is created by some Customer)

The suggestic e information about the DL-Learner plugir .

Figure 7. Screenshot of the result table of the DL-Learner plugin in OntoWiki.

gestions for learning the class “customer requirement” are shown in Manchester OWL Syntax. Similar to the Protégé plugin, the user is presented a table of suggestions along with their accuracy value. Additional details about the instances of “customer requirement” covered by a suggested class expressions and additionally contained instances can be viewed via a toggle button. The modular design of OntoWiki allows rich user interaction: Each resource, e.g. a class, property, or individual, can be viewed and subsequently modified directly from the result table as shown for “design requirement” in the screenshot. For instance, a knowledge engineer could decide to import additional information available as Linked Data and run the CELOE algorithm again to see whether different suggestions are provided with additional background knowledge.

7. Conclusions

Ontology construction may be a burdensome and time consuming task. To cope with this problem, the usage of machine learning techniques has been proposed. Specifically, the problem is regarded as a (supervised) concept learning problem where, given a set of individual resources labeled as instances of a target concept, the goal is to find an intensional concept description for them. Particularly, from those labels, axioms can be induced, which can then be confirmed by the knowledge engineer. The concept learning problem is tackled as a search through a space of candidate descriptions in the reference representation guided by exemplars of the target concepts. Those techniques are also applicable in other domains, e.g. question answering [28]. After surveyed existing methods and some basics on refinement operators, three different algorithms have been presented and compared in more detail: DL-FOIL, CELOE and TERMITIS.

References

[1] Sören Auer, Sebastian Dietzold, Jens Lehmann, and Thomas Riechert. OntoWiki: A tool for social, semantic collaboration. In Natalya Fridman Noy, Harith Alani, Gerd Stumme, Peter Mika, York Sure, and

Denny Vrandecic, editors, *Proceedings of the Workshop on Social and Collaborative Construction of Structured Knowledge (CKC 2007) at the 16th International World Wide Web Conference (WWW2007) Banff, Canada, May 8, 2007*, volume 273 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[2] Sören Auer, Sebastian Dietzold, and Thomas Riechert. Ontowiki - a tool for social, semantic collaboration. In *ISWC 2006*, volume 4273 of *LNCS*, pages 736–749. Springer, 2006.

[3] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniel Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[4] Liviu Badea and Shan hwei Nienhuys-Cheng. A refinement operator for description logics. In *Proc. of the Int. Conf. on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 40–59. Springer, 2000.

[5] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

[6] Henrik Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proc. of the Int. Joint Conf. on Artificial Intelligence, IJCAI95*, pages 1194–1200. Morgan Kaufmann, 1995.

[7] Lorenz Bühmann and Jens Lehmann. Universal OWL axiom enrichment for large knowledge bases. In *Proceedings of EKAW 2012*, 2012.

[8] Lorenz Buhmann and Jens Lehmann. Pattern based knowledge base enrichment. In *12th International Semantic Web Conference, 21-25 October 2013, Sydney, Australia*, 2013.

[9] William W. Cohen and Haym Hirsh. Learnability of description logics. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*. ACM Press, 1992.

[10] William W. Cohen and Haym Hirsh. Learning the CLASSIC description logic. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 121–133. Morgan Kaufmann, 1994.

[11] Floriana Esposito, Nicola Fanizzi, Luigi Iannone, Ignazio Palmisano, and Giovanni Semeraro. Knowledge-intensive induction of terminologies from metadata. In *The Semantic Web – ISWC 2004: Third International Semantic Web Conference. Proceedings*, pages 441–455. Springer, 2004.

[12] Nicola Fanizzi, Claudia d'Amato, and Floriana Esposito. DL-FOIL: Concept learning in description logics. In F. Zelezný and N. Lavrac, editors, *Proceedings of the 18th International Conference on Inductive Logic Programming, ILP2008*, volume 5194 of *LNAI*, pages 107–121. Springer, 2008.

[13] Nicola Fanizzi, Claudia d'Amato, and Floriana Esposito. Induction of concepts in web ontologies through terminological decision trees. In José L. Balcázar et al., editors, *Proceedings of ECML PKDD 2010, Part I*, volume 6321 of *LNCS/LNAI*, pages 442–457. Springer, 2010.

[14] Nicola Fanizzi, Floriana Esposito, Stefano Ferilli, and Giovanni Semeraro. A methodology for the induction of ontological knowledge from semantic annotations. In *Proc. of the Conf. of the Italian Association for Artificial Intelligence*, volume 2829 of *LNAI/LNCS*, pages 65–77. Springer, 2003.

[15] Nicola Fanizzi, Stefano Ferilli, Luigi Iannone, Ignazio Palmisano, and Giovanni Semeraro. Downward refinement in the $\mathcal{ALN}$ description logic. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems, HIS2004*, pages 68–73. IEEE Computer Society, 2005.

[16] Sally A. Goldman, Stephen S. Kwek, and Stephen D. Scott. Learning from examples with unspecified attribute values. *Information and Computation*, 180(2):82–100, 2003.

[17] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of OWL class descriptions on very large knowledge bases. *International Journal on Semantic Web and Information Systems*, 5(2):25–48, 2009.

[18] Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of owl class expressions on very large knowledge bases and its applications. In Interoperability Semantic Services and Web Applications: Emerging Concepts, editors, *Learning of OWL Class Expressions on Very Large Knowledge Bases and its Applications*, chapter 5, pages 104–130. IGI Global, 2011.

[19] Luigi Iannone, Ignazio Palmisano, and Nicola Fanizzi. An algorithm based on counterfactuals for concept learning in the semantic web. *Applied Intelligence*, 26(2):139–159, 2007.

[20] Josué Iglesias and Jens Lehmann. Towards integrating fuzzy logic capabilities into an ontology-based inductive logic programming framework. In *Proc. of the 11th International Conference on Intelligent Systems Design and Applications (ISDA)*, 2011.

[21] Nobuhiro Inuzuka, Masakage Kamo, Naohiro Ishii, Hirohisa Seki, and Hidenori Itoh. Tow-down induction of logic programs from incomplete samples. In *Selected Papers from the 6th International Workshop on Inductive Logic Programming, ILP96*, volume 1314 of *LNAI*, pages 265–282. Springer, 1997.

[22] Jörg Uwe Kietz and Katharina Morik. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, 14(2):193–218, 1994.

[23] Agnieszka Ławrynowicz and Volker Tresp. Introducing machine learning. In Jens Lehmann and Johanna Völker, editors, *Perspectives on Ontology Learning*, Studies on the Semantic Web. AKA Heidelberg /

IOS Press, 2014.

[24] Jens Lehmann. Hybrid learning of ontology classes. In *Proc. of the 5th Int. Conference on Machine Learning and Data Mining MLDM*, volume 4571 of *Lecture Notes in Computer Science*, pages 883–898. Springer, 2007.

[25] Jens Lehmann. DL-Learner: learning concepts in description logics. *Journal of Machine Learning Research (JMLR)*, 10:2639–2642, 2009.

[26] Jens Lehmann. *Learning OWL Class Expressions*. PhD thesis, University of Leipzig, 2010. PhD in Computer Science.

[27] Jens Lehmann, Sören Auer, Lorenz Bühmann, and Sebastian Tramp. Class expression learning for ontology engineering. *Journal of Web Semantics*, 9:71 – 81, 2011.

[28] Jens Lehmann and Lorenz Bühmann. Autosparql: Let users query your knowledge base. In *Proceedings of ESWC 2011*, 2011.

[29] Jens Lehmann and Christoph Haase. Ideal downward refinement in the el description logic. In *Proc. of the Int. Conf. on Inductive Logic Programming*, volume 5989 of *LNCS*, pages 73–87. Springer, 2009.

[30] Jens Lehmann and Pascal Hitzler. Foundations of refinement operators for description logics. In *ILP 2007*, volume 4894 of *LNCS*, pages 161–174. Springer, 2008.

[31] Jens Lehmann and Pascal Hitzler. A refinement operator based learning algorithm for the ALC description logic. In *ILP 2007*, volume 4894 of *LNCS*, pages 147–160. Springer, 2008.

[32] Jens Lehmann and Pascal Hitzler. Concept learning in description logics using refinement operators. *Machine Learning journal*, 78(1-2):203–250, 2010.

[33] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[34] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[35] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[36] Luc De Raedt. *Logical and Relational Learning*. Springer, 2008.

[37] Thomas Riechert, Kim Lauenroth, Jens Lehmann, and Sören Auer. Towards semantic based requirements engineering. In *I-KNOW 2007*, 2007.

[38] Gunnar Teege. A subtraction operation for description logics. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 540–550. Morgan Kaufmann, 1994.